Desconocido

## 14 Desarrollando el Juego del Sr. Hombre de Palo



Ahora que hemos creado las imágenes del Sr. Hombre de Palo Corre *por la Salida* , podemos empezar a desarrollar el código. La descripción del juego en el capítulo anterior nos da una idea de lo que necesitaremos: una figura de palo que pueda correr y saltar y plataformas a las que deba saltar. Escribiremos código para mostrar la figura de palo y moverla por la pantalla, así como para mostrar las plataformas. Pero antes de escribir este código, tenemos que crear el lienzo para mostrar nuestra imagen de fondo.

### Crear la clase del juego

En primer lugar, crearemos una clase llamada Game que será el controlador principal de nuestro programa. La clase Game tendrá una función \_\_init\_\_ para inicializar el juego y una función mainloop para realizar la animación.

### Establecer el título de la ventana y crear el lienzo

En la primera parte de la función \_\_init\_\_ , estableceremos el título de la ventana y crearemos el lienzo. Como verás, esta parte del código es similar al código que escribimos para *¡Botar!* en [el Capítulo 11 .](ch11.xhtml#ch11) Abre un nuevo archivo en IDLE e introduce el siguiente código, y luego guarda tu archivo como *stickmangame.py .* Asegúrate de guardarlo en la carpeta *stickman* que creamos en el [Capítulo 13 :](ch13.xhtml#ch13)

from tkinter import \*  
import random  
import time   
  
class Game:  
 def \_\_init\_\_(self):  
 self.tk = Tk()  
 self.tk.title('Mr. Stick Man Races for the Exit')  
 self.tk.resizable(0, 0)  
 self.tk.wm\_attributes('-topmost', 1)  
 self.canvas = Canvas(self.tk, width=500, height=500,  
 highlightthickness=0)  
 self.canvas.pack()  
 self.tk.update()  
 self.canvas\_height = self.canvas.winfo\_height()  
 self.canvas\_width = self.canvas.winfo\_width()

En la primera mitad de este programa (las líneas que van de tkinter import \* a self.tk.wm\_attributes ), creamos el objeto tk y luego establecemos el título de la ventana con self.tk.title a ("Mr. Stick Man Races for the Exit"). Hacemos que la ventana *sea fija* (para que no pueda cambiar de tamaño) llamando a la función resizable , y luego movemos la ventana delante de todas las demás ventanas con la función wm\_attributes .

A continuación, creamos el lienzo con la línea self.canvas = Canvas , y llamamos a las funciones pack y update del objeto tk . Por último, creamos dos variables para nuestra clase Game , height y width , para almacenar la altura y la anchura (utilizamos las funciones winfo\_height y winfo\_width para obtener el tamaño del lienzo).

Nota

*La barra invertida (\) en la línea self.canvas = Canvas se utiliza sólo para separar una línea larga de código. No es necesaria en este caso, pero la he incluido aquí para facilitar la lectura, ya que la línea entera no cabe en la página.*

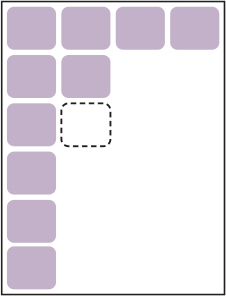
### Finalizando la función \_\_init\_\_

Ahora introduce el resto de la función \_\_init\_\_ en el archivo *stickman game.* py que acabas de crear. Este código cargará la imagen de fondo y luego la mostrará en el lienzo:

self.tk.update()  
 self.canvas\_height = self.canvas.winfo\_height()  
 self.canvas\_width = self.canvas.winfo\_width()  
 self.bg = PhotoImage(file='background.gif')  
 w = self.bg.width()  
 h = self.bg.height()  
 ➊ for x in range(0, 5):  
 ➋ for y in range(0, 5):  
 self.canvas.create\_image(x \* w, y \* h,   
 image=self.bg, anchor='nw')  
 self.sprites = []  
 self.running = True

En la línea que comienza self.bg , creamos la variable bg , que contiene un objeto PhotoImage -el archivo de imagen de fondo llamado *background.gif* que creamos en el [Capítulo 13](ch13.xhtml#ch13) en la [página 210 .](ch13.xhtml#ch13lev1sec8) A continuación, almacenamos la anchura y la altura de la imagen en las variables w y h . Las funciones de clase PhotoImage width y height devuelven el tamaño de la imagen una vez cargada.

A continuación vienen dos bucles dentro de esta función. Para entender lo que hacen, imagina que tienes un pequeño sello de caucho cuadrado, una almohadilla de tinta y un gran trozo de papel. ¿Cómo puedes utilizar el sello para llenar el papel de cuadrados de colores? Bueno, podrías cubrir la página con sellos al azar hasta llenarla. El resultado sería un desastre, y tardaría un rato en completarse,  pero llenaría la página. O podrías empezar a estampar hacia abajo en una columna y luego volver a la parte superior y empezar a estampar hacia abajo en la página de la siguiente columna, como se muestra en la [Figura 14-1 .](ch14.xhtml#ch14fig01)



Figura*14-1* : Estampar hacia abajo en la página

La imagen de fondo que creamos en el capítulo anterior es nuestro sello. Sabemos que el lienzo mide 500 píxeles de ancho y 500 píxeles de ancho, y que hemos creado una imagen de fondo de 100 píxeles cuadrados. Esto nos dice que necesitamos cinco columnas a lo ancho y cinco filas hacia abajo para llenar la pantalla de imágenes. Utilizamos un bucle for ➊ para calcular las columnas a lo ancho, y otro bucle for ➋ para calcular las filas hacia abajo.

A continuación, multiplicamos la primera variable del bucle x por la anchura de la imagen ( x \* w ) para determinar hasta dónde vamos a dibujar a lo ancho, y luego multiplicamos la segunda variable del bucle y por la altura de la imagen ( y \* h ) para calcular hasta dónde vamos a dibujar hacia abajo. Utilizamos la función create\_image del objeto canvas ( self.canvas.create \_image ) para dibujar la imagen en la pantalla utilizando esas coordenadas.

Por último, creamos las variables sprites , que contiene una lista vacía, y running , que contiene el valor booleano True . Utilizaremos estas variables más adelante en el código de nuestro juego.

### Creación de la función mainloop

Utilizaremos la función mainloop de la clase Game para animar nuestro juego. Esta función se parece mucho al bucle principal (o bucle de animación) que creamos para el juego *¡Rebota!* en [el Capítulo 11 .](ch11.xhtml#ch11)

Nuestra función es la siguiente:

for x in range(0, 5):  
 for y in range(0, 5):  
 self.canvas.create\_image(x \* w, y \* h,   
 image=self.bg, anchor='nw')  
 self.sprites = []  
 self.running = True  
   
 def mainloop(self):  
 while True:  
 if self.running == True:  
 for sprite in self.sprites:  
 sprite.move()  
 self.tk.update\_idletasks()  
 self.tk.update()  
 time.sleep(0.01)

Creamos un bucle while que se ejecutará hasta que se cierre la ventana del juego ( while True es un bucle infinito, que vimos por primera vez en la [página 175](ch11.xhtml#para95) ). A continuación, comprobamos si la variable running es igual a True . Si lo es, recorremos todos los sprites de la lista de sprites ( self.sprites ), llamando a la función move para cada uno de ellos. (Aún no hemos creado ningún sprites, así que este código no hará nada si ejecutamos el programa ahora, pero será útil más adelante).

Las tres últimas líneas de la función obligan al objeto tk a redibujar la pantalla y a dormir durante una fracción de segundo, como hicimos con el juego *¡Bounce!*

Así que puedes ejecutar este código, añadir las dos líneas siguientes (observa que no hay sangría) y guardar el archivo:



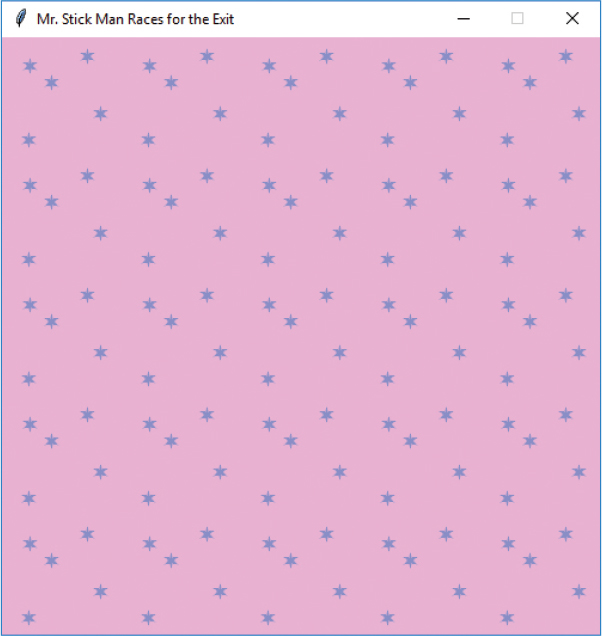
g = Game()  
g.mainloop()

Nota

*Asegúrate de añadir este código al final del archivo de tu juego.* Asegúrate*también de que tus imágenes y el archivo Python están todos en la* *carpeta* stickman *que creaste en* el [*Capítulo 13 .*](ch13.xhtml#ch13)

Este código crea un objeto de la clase Game y lo guarda como variable g . A continuación, llamamos a la función mainloop sobre el nuevo objeto para dibujar la pantalla.

Una vez guardado el programa, ejecútalo en IDLE con **Ejecutar** **▸ Ejecutar Módulo** . Debería aparecer una ventana con la imagen de fondo llenando el lienzo, como en la [Figura 14-2 .](ch14.xhtml#ch14fig02)



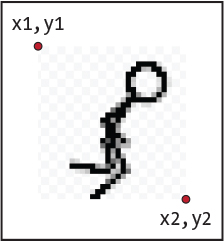
Figura*14-2: Fondo del juego*

Con esto, hemos añadido un bonito fondo para nuestro juego y creado un bucle de animación que dibujará los sprites por nosotros (una vez que los hayamos creado).

### Crear la clase Coords

Ahora crearemos la clase que utilizaremos para especificar la posición de los sprites en la pantalla de nuestro juego. Esta clase almacenará las coordenadas superior izquierda ( *x1* e *y1* ) e inferior derecha ( *x2* e *y2* ) de cualquier componente de nuestro juego.

[La Figura 14-3](ch14.xhtml#ch14fig03) muestra cómo podrías registrar la posición de la imagen de la figura de palo utilizando estas coordenadas.



*Figura 14-3: Dónde se encuentran las coordenadas x e y en la* figura de palo

Nuestra nueva clase, Coords , sólo contendrá una función \_\_init\_\_ , a la que pasaremos los cuatro parámetros ( x1 , y1 , x2 y y2 ). Pon este código al principio del archivo *stickmangame.py* :

class Coords:  
 def \_\_init\_\_(self, x1=0, y1=0, x2=0, y2=0):  
 self.x1 = x1  
 self.y1 = y1  
 self.x2 = x2  
 self.y2 = y2

Observa que cada parámetro se guarda como una variable objeto del mismo nombre ( x1 , y1 , x2 , y y2 ). En breve utilizaremos objetos de esta clase.

### Comprobación de colisiones

Una vez que sabemos cómo almacenar la posición de los sprites de nuestro juego, necesitamos una forma de saber si un sprite ha colisionado con otro, como cuando el Sr. Hombre de Palo salta por la pantalla y choca con una de las plataformas. Para que este problema sea más fácil de resolver, podemos dividirlo en dos problemas más pequeños: comprobar si los sprites colisionan verticalmente y comprobar si los sprites colisionan horizontalmente. Luego podemos combinar nuestras soluciones para ver si dos sprites chocan en cualquier dirección.

### Colisión horizontal de sprites

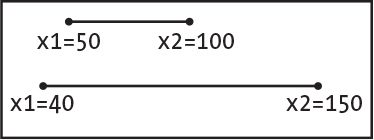
En primer lugar, crearemos la función within\_x para determinar si un conjunto de *coordenadas* *x* ( *x1* y *x2* ) se ha cruzado con otro conjunto de *coordenadas x* ( de nuevo, *x1* y *x2* ). Añade lo siguiente directamente debajo de la clase Coords :

class Coords:  
 def \_\_init\_\_(self, x1=0, y1=0, x2=0, y2=0):  
 self.x1 = x1  
 self.y1 = y1  
 self.x2 = x2  
 self.y2 = y2  
  
def within\_x(co1, co2):  
 ➊ if co1.x1 > co2.x1 and co1.x1 < co2.x2:  
 return True  
 ➋ elif co1.x2 > co2.x1 and co1.x2 < co2.x2:  
 return True  
 elif co2.x1 > co1.x1 and co2.x1 < co1.x2:  
 return True  
 elif co2.x2 > co1.x1 and co2.x2 < co1.x2:  
 return True  
 else:  
 return False

La función within\_x toma los parámetros co1 y co2 , ambos objetos Coords . Primero comprobamos si la posición más a la izquierda del primer objeto de coordenadas ( co1.x1 ) está entre la posición más a la izquierda ( co2.x1 ) y la posición más a la derecha ( co2.x2 ) del segundo objeto de coordenadas ➊ . Si es así, devolvemos True .



Echemos un vistazo a dos líneas con *coordenadas x* superpuestas para entender cómo funciona esto. Cada línea de la [Figura 14-4](ch14.xhtml#ch14fig04) comienza en x1 y termina en x2 .



*Figura 14-4: Coordenadas horizontales (* x) superpuestas

La primera línea de este diagrama ( co1 ) comienza en la posición de píxel 50 ( x1 ) y termina en 100 ( x2 ). La segunda línea ( co2 ) comienza en la posición 40 y termina en 150. En este caso, como la posición x1 de la primera línea está entre las posiciones x1 y x2 de la segunda línea, la afirmación if de la función sería verdadera para estos dos conjuntos de coordenadas.

Con la primera sentencia elif ➋ , vemos si la posición más a la derecha de la primera línea ( co1.x2 ) está entre la posición más a la izquierda ( co2.x1 ) y la posición más a la derecha ( co2.x2 ) de la segunda. Si es así, devolvemos True . Las dos siguientes sentencias elif hacen casi lo mismo: comprueban las posiciones más a la izquierda y más a la derecha de la segunda línea ( co2 ) con la primera ( co1 ).

Si ninguna de las sentencias if coincide, llegamos a else y devolvemos False . Esto equivale a decir: "No, los dos objetos de coordenadas no se cruzan horizontalmente".

Para ver un ejemplo del funcionamiento de la función, vuelve a mirar la [Figura 14-4 .](ch14.xhtml#ch14fig04) Las posiciones x1 y x2 del primer objeto de coordenadas son 50 y 100, y las posiciones x1 y x2 del segundo objeto de coordenadas son 40 y 150. Esto es lo que ocurre cuando llamamos a la función within\_x que hemos creado:

>>> c1 = Coords(50, 50, 100, 100)  
>>> c2 = Coords(40, 40, 150, 150)  
>>> print(within\_x(c1, c2))  
True

La función devuelve True . Este es el primer paso para determinar si un sprite ha chocado con otro. Por ejemplo, cuando creemos una clase para el Sr. Hombre de Palo y para las plataformas, podremos saber si sus *coordenadas x* se han cruzado.

No es una buena práctica tener muchas sentencias if o elif que devuelvan el mismo valor. Para resolver este problema, podemos acortar la función within\_x rodeando cada una de sus condiciones con paréntesis, separados por la palabra clave or . Para obtener una función algo más ordenada y con menos líneas de código, puedes cambiar la función  para que tenga este aspecto:

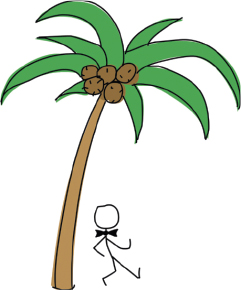
def within\_x(co1, co2):  
 if (co1.x1 > co2.x1 and co1.x1 < co2.x2) \  
 or (co1.x2 > co2.x1 and co1.x2 < co2.x2) \  
 or (co2.x1 > co1.x1 and co2.x1 < co1.x2) \  
 or (co2.x2 > co1.x1 and co2.x2 < co1.x2):  
 return True  
 else:  
 return False

Para extender la sentencia if a lo largo de varias líneas, de modo que no acabemos con una línea realmente larga que contenga todas las condiciones, utilizamos una barra invertida (\), como se muestra arriba.

### Sprites que colisionan verticalmente

También necesitamos saber si los sprites colisionan verticalmente. La función within\_y es muy similar a la función within\_x . Para crearla, comprobamos si la posición *y1* de la primera coordenada se ha cruzado con las posiciones *y1* e *y2* de la segunda, y viceversa.

Añade la siguiente función debajo de la función within\_x . Esta vez, utilizaremos la versión más corta del código (en lugar de montones de sentencias if ):



def within\_y(co1, co2):  
 if (co1.y1 > co2.y1 and co1.y1 < co2.y2) \  
 or (co1.y2 > co2.y1 and co1.y2 < co2.y2) \  
 or (co2.y1 > co1.y1 and co2.y1 < co1.y2) \  
 or (co2.y2 > co1.y1 and co2.y2 < co1.y2):  
 return True  
 else:  
 return False

Nuestras funciones within\_x y within\_y se parecen bastante porque, en el fondo, están haciendo cosas parecidas.

### Ponerlo todo junto: Nuestro código final de detección de colisiones

Una vez que hemos determinado si un conjunto de *coordenadas x* se ha cruzado con otro, y hemos hecho lo mismo con *las coordenadas y* , podemos escribir funciones para ver si un sprite ha chocado con otro sprite y por qué lado. Haremos esto con las funciones collided\_left , collided\_right , collided\_top , y collided\_bottom .

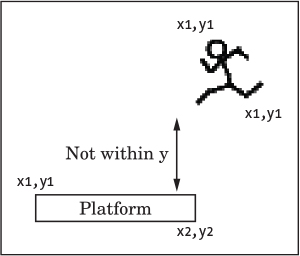
#### La función colisionado\_izquierdo

Añade el siguiente código para la función collided\_left debajo de las dos funciones within que acabamos de crear:

def collided\_left(co1, co2):  
 if within\_y(co1, co2):  
 if co1.x1 >= co2.x1 and co1.x1 <= co2.x2:  
 return True  
 return False

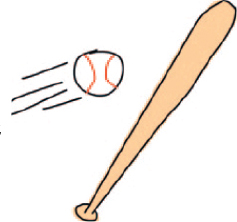
Esta función nos dice si el lado izquierdo (el valor x1 ) de un primer objeto de coordenadas ha chocado con otro objeto de coordenadas.

La función toma dos parámetros: co1 (el primer objeto de coordenadas) y co2 (el segundo objeto de coordenadas). Comprobamos si los dos objetos de coordenadas se han cruzado verticalmente, utilizando la función within\_y . Al fin y al cabo, no tiene sentido comprobar si el Sr. Hombre de Palo ha chocado contra una plataforma si está flotando muy por encima de ella (como en [la Figura 14-5](ch14.xhtml#ch14fig05) ).



*Figura 14-5:* El Sr. Hombre de Palo*por encima* de la plataforma

A continuación, comprobamos si el valor de la posición más a la izquierda del primer objeto de coordenadas ( co1.x1 ) ha chocado con la posición x2 del segundo objeto de coordenadas ( co2.x2 ). Si es así, debe ser menor o igual que la posición x2 . También comprobamos que no ha sobrepasado la posición x1 . Si la ha sobrepasado, devolvemos True . Si ninguna de las afirmaciones de if es cierta, devolvemos False .



#### La función derecha\_colisionada

La función collided\_right se parece mucho a collided\_left :

def collided\_right(co1, co2):  
 if within\_y(co1, co2):  
 if co1.x2 >= co2.x1 and co1.x2 <= co2.x2:  
 return True  
 return False

Al igual que con collided\_left , comprobamos si las *coordenadas y* se han cruzado entre sí, utilizando la función within\_y . A continuación, comprobamos si el valor x2 del primer objeto de coordenadas se encuentra entre las posiciones x1 y x2 del segundo objeto de coordenadas, y devolvemos True si es así. En caso contrario, devolvemos False .

#### La función collided\_top

La función collided\_top es muy similar a las dos funciones que acabamos de añadir:

def collided\_top(co1, co2):  
 if within\_x(co1, co2):  
 if co1.y1 >= co2.y1 and co1.y1 <= co2.y2:  
 return True  
 return False

Esta vez, comprobamos si las coordenadas se han cruzado horizontalmente, utilizando la función within\_x . A continuación, comprobamos si la posición más alta de la primera coordenada ( co1.y1 ) se ha cruzado con la posición y2 de la segunda coordenada, pero no con su posición y1  . Si es así, devolvemos True (la parte superior de la primera coordenada ha chocado con la segunda coordenada).

#### La función colided\_bottom

Nuestra última función, collided\_bottom , es un poco diferente:

def collided\_bottom(y, co1, co2):  
 if within\_x(co1, co2):  
 y\_calc = co1.y2 + y  
 ➊ if y\_calc >= co2.y1 and y\_calc <= co2.y2:  
 return True  
 return False

Esta función toma un parámetro adicional, y , un valor que añadimos a la posición y de la primera coordenada. Nuestra sentencia if comprueba si las coordenadas se han cruzado horizontalmente (como hicimos con collided\_top ). A continuación, añadimos el valor del parámetro y a la posición y2 de la primera coordenada, y almacenamos el resultado en la variable y\_calc . Si el nuevo valor calculado se encuentra entre los valores y1 y y2 de la segunda coordenada ➊ , devolvemos True porque la parte inferior de la coordenada co1 ha tocado la parte superior de la coordenada co2 . Sin embargo, si ninguna de las afirmaciones if es cierta, devolvemos False .

Necesitamos el parámetro adicional y porque el Sr. Hombre de Palo podría caerse de una plataforma. A diferencia de las otras funciones collided , necesitamos poder comprobar *si* colisionaría en la parte inferior, en lugar de si ya lo ha hecho. Si se cae de una plataforma y sigue flotando en el aire, nuestro juego no será muy realista; por eso, mientras camina, comprobamos si ha colisionado con algo a la izquierda o a la derecha. Cuando comprobamos debajo de él, vemos si colisionaría con la plataforma; si no, ¡tiene que caer estrepitosamente!

### Creación de la clase Sprite

La clase padre de nuestros elementos de juego, Sprite , proporcionará dos funciones: move para mover el sprite, y coords para devolver la posición actual del sprite en la pantalla. Añadimos el código de la clase Sprite debajo de la función collided\_bottom , como sigue:

class Sprite:  
 def \_\_init\_\_(self, game):  
 self.game = game  
 self.endgame = False  
 self.coordinates = None  
  
 def move(self):  
 pass  
  
 def coords(self):  
 return self.coordinates

La función \_\_init\_\_ de la clase Sprite toma un único parámetro, game , que será el objeto del juego. Lo necesitamos para que cualquier sprite que creemos pueda acceder a la lista de otros sprites del juego. Almacenamos el parámetro del juego como una variable de objeto.

A continuación, almacenamos la variable de objeto endgame , que utilizaremos para indicar el final del juego. (En este momento, se establece en False .) La última variable de objeto, coordinates , se establece en nada ( None ).

La función move no hace nada en esta clase padre, así que utilizamos la palabra clave pass en el cuerpo de esta función. La función coords simplemente devuelve la variable de objeto coordinates .

Así que nuestra clase Sprite tiene una función move que no hace nada y una función coords que no devuelve ninguna coordenada. No parece muy útil, ¿verdad? Sin embargo, cualquier clase que tenga como padre a Sprite siempre tendrá las funciones move y coords . Así, en el bucle principal del juego, cuando recorramos una lista de sprites, llamar a la función move no provocará ningún error porque cada sprite tiene esa función.



Nota

*Las clases con funciones que no hacen gran cosa son bastante comunes en programación.* En*cierto modo, son un acuerdo que garantiza que todos los hijos de una clase proporcionan el mismo tipo de funcionalidad, aunque en algunos casos las funciones de las clases hijas no hagan nada.*

### Añadir las plataformas

Ahora añadiremos las plataformas. Nuestra clase para objetos plataforma, PlatformSprite , será una clase hija de Sprite . La función \_\_init\_\_ de esta clase tomará un parámetro game (como hace la clase padre Sprite ), así como una imagen, las posiciones x y y , y la imagen width y height . Aquí tienes el código de la clase PlatformSprite , que va directamente debajo de la clase Sprite :

class PlatformSprite(Sprite):  
 def \_\_init\_\_(self, game, photo\_image, x, y, width, height):  
 Sprite.\_\_init\_\_(self, game)  
 self.photo\_image = photo\_image  
 self.image = game.canvas.create\_image(x, y,   
 image=self.photo\_image, anchor='nw')  
 self.coordinates = Coords(x, y, x + width, y + height)

Cuando definimos la clase PlatformSprite , le damos un único parámetro: el nombre de la clase padre ( Sprite ). La función \_\_init\_\_ tiene siete parámetros: self , game , photo\_image , x , y , width y height .

Llamamos a la función \_\_init\_\_ de la clase padre, Sprite , utilizando self y game como valores de los parámetros, porque aparte del parámetro self , la función \_\_init\_\_ de la clase Sprite sólo toma un parámetro: game .

En este punto, si creáramos un objeto PlatformSprite , tendría todas las variables de objeto de su clase padre ( game , endgame y coordinates ), simplemente porque hemos llamado a la función \_\_init\_\_ en Sprite .



A continuación, guardamos el parámetro photo\_image como variable de objeto, y utilizamos la variable canvas del objeto juego para dibujar la imagen en pantalla con create\_image .

Por último, creamos un objeto Coords con los parámetros x y y como dos primeros argumentos. A continuación, añadimos los parámetros width y height a estos parámetros para los dos segundos argumentos.

Aunque la variable coordinates está configurada como None en la clase padre Sprite , la hemos cambiado en nuestra clase hija PlatformSprite por un objeto Coords real, que contiene la ubicación en vivo de la imagen de la plataforma en la pantalla.

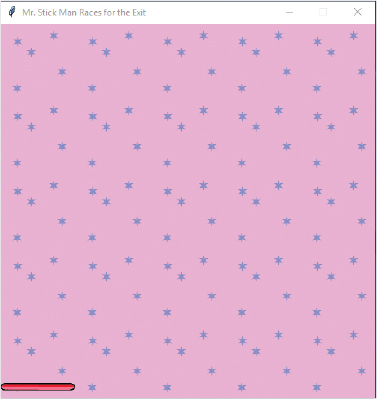
### Añadir un objeto Plataforma

Vamos a añadir una plataforma al juego para ver cómo queda. Cambia las dos últimas líneas del archivo del juego ( *stickmangame.py* ):

g = Game()  
➊ platform1 = PlatformSprite(g, PhotoImage(file='platform1.gif'),   
 0, 480, 100, 10)  
➋ g.sprites.append(platform1)  
 g.mainloop()

Creamos un objeto de la clase PlatformSprite , pasándole la variable de nuestro juego ( g ), junto con un objeto PhotoImage (que utiliza la primera de nuestras imágenes de plataforma, *plataforma1.gif* ) ➊ . También le pasamos la posición en la que queremos dibujar la plataforma (0 píxeles de ancho y 480 píxeles de bajo, cerca de la parte inferior del lienzo), junto con la altura y la anchura de nuestra imagen (100 píxeles de ancho y 10 píxeles de alto). Añadimos este sprite a la lista de sprites de nuestro objeto de juego ➋ .

Si ejecutas ahora el juego, deberías ver una plataforma en la parte inferior izquierda de la pantalla, como en [la Figura 14-6 .](ch14.xhtml#ch14fig06)



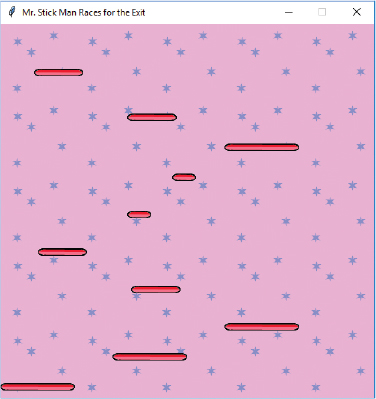
*Figura 14-6: Visualización* de una plataforma

### Añadir un montón de plataformas

Vamos a añadir un montón de plataformas. Cada plataforma tendrá diferentes *posiciones* *x e* *y* , por lo que estarán dispersas por la pantalla. Utiliza el siguiente código:

g = Game()  
platform1 = PlatformSprite(g, PhotoImage(file='platform1.gif'),   
 0, 480, 100, 10)  
platform2 = PlatformSprite(g, PhotoImage(file='platform1.gif'),   
 150, 440, 100, 10)  
platform3 = PlatformSprite(g, PhotoImage(file='platform1.gif'),   
 300, 400, 100, 10)  
platform4 = PlatformSprite(g, PhotoImage(file='platform1.gif'),   
 300, 160, 100, 10)  
platform5 = PlatformSprite(g, PhotoImage(file='platform2.gif'),   
 175, 350, 66, 10)  
platform6 = PlatformSprite(g, PhotoImage(file='platform2.gif'),   
 50, 300, 66, 10)  
platform7 = PlatformSprite(g, PhotoImage(file='platform2.gif'),   
 170, 120, 66, 10)  
platform8 = PlatformSprite(g, PhotoImage(file='platform2.gif'),   
 45, 60, 66, 10)  
platform9 = PlatformSprite(g, PhotoImage(file='platform3.gif'),   
 170, 250, 32, 10)  
platform10 = PlatformSprite(g, PhotoImage(file='platform3.gif'),   
 230, 200, 32, 10)  
g.sprites.append(platform1)  
g.sprites.append(platform2)  
g.sprites.append(platform3)  
g.sprites.append(platform4)  
g.sprites.append(platform5)  
g.sprites.append(platform6)  
g.sprites.append(platform7)  
g.sprites.append(platform8)  
g.sprites.append(platform9)  
g.sprites.append(platform10)  
g.mainloop()

Primero creamos un montón de objetos PlatformSprite , guardándolos como variables platform1 , platform2 , platform3 , y así sucesivamente, hasta platform10 . A continuación, añadimos cada plataforma a la variable sprites , que creamos  en nuestra clase Game . Si ejecutas ahora el juego, debería parecerse a [la Figura 14-7 .](ch14.xhtml#ch14fig07)



*Figura 14-7: Visualización de todas las* plataformas

¡Hemos creado lo básico de nuestro juego! Ahora estamos listos para añadir a nuestro personaje principal, el Sr. Hombre de Palo.

### Lo que has aprendido

En este capítulo, creaste la clase Game y dibujaste la imagen de fondo en la pantalla. Aprendiste a determinar si una posición horizontal o vertical está dentro de los límites de otras dos posiciones horizontales o verticales creando las funciones within\_x y within\_y . A continuación, utilizaste estas funciones para crear nuevas funciones que determinan si un objeto de coordenadas ha colisionado con otro. Utilizaremos estas funciones en los próximos capítulos cuando animemos al Sr. Hombre de Palo y necesitemos detectar si ha colisionado con una plataforma mientras se mueve por el lienzo.

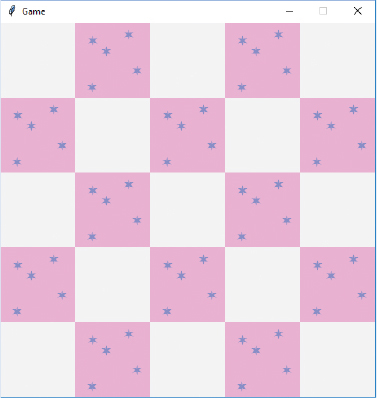
También creamos una clase padre Sprite y su primera clase hija, PlatformSprite , que utilizamos para dibujar las plataformas en el lienzo.

### Puzles de programación

Los siguientes puzzles de programación son algunas formas de experimentar con la imagen de fondo del juego. Comprueba tus respuestas en http://python-for-kids.com [*.*](http://python-for-kids.com)

#### #nº 1: Tablero de ajedrez

Prueba a cambiar la clase Game para que la imagen de fondo se dibuje como un tablero de damas, como en la [Figura 14-8 .](ch14.xhtml#ch14fig08)



*Figura 14-8* : Fondo como un tablero de damas

#### #2: Tablero de damas de dos imágenes

Una vez que hayas descubierto cómo crear un efecto de damero, prueba a utilizar dos imágenes alternadas. Inventa otra imagen de fondo de pantalla (utilizando tu programa gráfico), y luego cambia la clase Game para que muestre un damero con dos imágenes alternadas en lugar de una imagen y el fondo en blanco.

#### #3: Estantería y lámpara

Puedes crear diferentes imágenes de fondo para hacer más interesante el fondo del juego. Crea una copia de la imagen de fondo; luego dibuja una estantería sencilla, una mesa con una lámpara o una ventana. Distribuye estas imágenes por la pantalla cambiando la clase Game para que muestre varias imágenes de fondo de pantalla diferentes.

#### #4: Fondo aleatorio

Como alternativa al tablero de ajedrez de dos imágenes, prueba a crear cinco imágenes de fondo diferentes. Puedes dibujarlas como un patrón repetido de imágenes de fondo (1, 2, 3, 4, 5, 1, 2, 3, 4, 5, etc.), o puedes dibujarlas aleatoriamente.

Sugerencia: Si importas el módulo random y colocas tus imágenes en una lista, prueba a utilizar random.choice() para elegir una al azar.